1-1-2013

Synergistically Coupling Of Solid State Drives And Hard Disks For Qos-Aware Virtual Memory

Ke Liu
Wayne State University,

SYNERGISTICALLY COUPLING OF SOLID STATE DRIVES AND HARD DISKS FOR QOS-AWARE VIRTUAL MEMORY

by

KE LIU

THESIS

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

2013

MAJOR: COMPUTER ENGINEERING

Approved by:

_____

Advisor **Date**

_____

ACKNOWLEDGEMENTS

The two years study at Wayne State University is very interesting and full of challenges. This thesis could not be achieved without help from many people in my life. I am very grateful to my advisor, Dr. Song Jiang, for his guidance, support, encouragement all the time. His guidance and extensive knowledge help me get through all the hard times of this work. Thanks to Dr. Kei Davis, who provided me an internship and a fantastic trip at Los Alamos National Laboratory, as his insightful comments on my work help it improved a lot. I would also like to thank the rest of my thesis committee, Prof. Cheng-zhong Xu and Prof. Weisong Shi, for their encouragement, suggestions, and hard questions to improve this work. I am very thankful to my friends Dr. Xuechen Zhang, Yuehai Xu, Jianqiang Ou, Dr. Jia Rao, Xiangping Bu, Kun Wang, Yudi Wei, who made weekly discussion with me and helped me a lot in daily life.

Lastly, I would like to thank my entire family, for their understanding, support and love for my over twenty years of study.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1
Introduction

Normally, desktops and laptops are statically configured with fixed size of dynamic random-access memory (DRAM). Although the price of DRAM per Gigabytes (GB) becomes increasingly lower and the size of DRAM on each desktop and laptop machine becomes larger and larger, the memory needs of applications grow dramatically too. The growing speed of memory demands for applications running on desktops and laptops is much faster than the increasing speed of DRAM size on desktops and laptops. When applications are demanding more memory than DRAM size, system would come into swapping. Typically, hard disks would be used as swapping devices and user experience might be deteriorated during swapping.

1.1 Using Solid State Drive as Swapping Device

Given the large performance gap between DRAM (memory) and hard disk, flash memory (flash)—with access latency much less than hard disk, with performance much less sensitive to random access than hard disk and with performance much better to sequential access than hard disk —has been widely employed to accelerate access of data normally stored on hard disk, or by itself to provide a fast storage system. Another use of flash is as an extension of memory, so that working sets of memory-intensive programs can overflow into the additional memory space [39, 12, 28, 33, 42]. This latter application is motivated by the exponential cost of DRAM as a function of density (e.g., ∼$10/GB for 2GB DIMM, ∼$200/GB for 8GB DIMM, and hundreds or even thousands of dollars per GB for DIMM over 64GB) and its high power consumption. In contrast, flash is much less expensive, has much greater density, and is much more energy efficient both in terms of energy consumption and needed cooling.

A typical approach for inclusion of flash in the memory system is to use it as swap space and map virtual memory onto it [39, 12, 28]. Much effort has been made to address the particular properties of flash for use in this context, such as access at page granularity, the write-before-erasure requirement, asymmetric read and write performance, and limited write cycles. The most challenging issue is flash's endurance—the limited number of write-erase cycles each flash block can perform before failure. A block of MLC (multi-level cell) NAND flash—the less expensive, higher density, and more widely used type—may only be erased and rewritten approximate 5000-10,000 times before it has unacceptably high bit error rates [23, 32]. Though SLC (single-level cell) flash can support a higher erasure limit (around 100,000) per flash block, increasing flash density can reduce this limit [24].

When the flash-based solid state drive is used for file storage, the limited number of write cycles that flash can endure may not be a significant issue because frequently accessed data is usually buffered in memory. However, if solid state drive is used as a memory extension with memory-intensive applications, the write rate to the solid state drive can be much higher and solid state drive longevity can be a serious concern. For example, for a program writing to its working set on a 64GB solid state drive of MLC flash at a sustained rate of 128MB/s, the slid state drive can become unreliable after about two months. Considering the write-amplification effect due to garbage collection at the flash translation layer (FTL), the lifetime could be reduced by a factor of up to 1000 [26]. To improve solid state drive's lifetime when used as swap space for processes' virtual memory, researchers have tried to reduce write traffic to solid state drive by using techniques such as minimizing the replacement of dirty pages [39], detecting pages containing all zero bytes to avoid swapping them [39], and managing swap space at the granularity of custom objects [12]. In these prior works, when solid state drive is proposed to serve as swap space, the conventional host, hard

disk, is excluded from consideration for use for the same purpose. This is in contrast to the scenario where solid state drive is used as file storage, wherein hard disk is often also used to provide large capacity and to reduce cost.

1.2 Thesis Contributions

In this thesis, we propose a page-swapping management scheme, *HybridSwap*, to reduce solid state drive writes in its use as a memory extension. In this scheme, we track page access patterns to identify pages of strong spatial locality to form page sequences and accordingly determine the destinations of pages to be swapped out. In this work, we make the following contributions.

- We propose to introduce the hard disk into the use of solid state drive for memory extension. We show that in representative memory-intensive applications, there is substantial sequential access that warrants the use of hard disk to significantly reduce solid state drive writes without significant performance loss compared to solid state drive-only swapping.

- We develop an efficient algorithm to record memory access history and to identify page access sequences and evaluate their locality. Swapping destinations are accordingly determined for the sequences to ensure that both high disk throughput and low solid state drive latency are exploited while high latency is avoided or hidden.

- We build a QoS-assurance mechanism into HybridSwap to demonstrate the flexibility of the system in bounding the performance penalty due to swapping. It allows users to specify a bound on the program stall time due to page faults as a percentage of the program's total run time.

- We have implemented the hybrid swapping system in the Linux kernel and have conducted extensive evaluation using representative benchmarks including key-value store for in-memory caching, image processing, and scientific computations. The results show that HybridSwap can reduce solid state drive swapping writes by 40% with performance comparable to that of using a solid state drive-only solution.

1.3 Thesis Organization

The rest of the dissertation is organized as follows:

Chapter 2 describes how virtual memory, especially paging virtual memory and swapping, works on modern computer architecture.

Chapter 3 first gives an overview about characteristics of hard disks and how existing Linux operating system making use of these characteristics for hard disks as its swapping device. Then we present the characteristics of solid state drives and the existing work on incorporating solid state drives for virtual memory.

Chapter 4 describes the design and implementation of HybridSwap which takes spatial and temporal locality into considerations for selecting pages to be swapped to an assigned destination (hard disks or solid state drives) with low overhead and high efficiency. Also, we take QoS-assurance mechanism into our design.

Chapter 5 describes the experimental setup, evaluates HybridSwap with representative benchmarks and analyzes the experiment results which showed that Hybrid could save writes to solid state drives without compromising the performance.

Chapter 6 summarizes our contributions and limitations in this work and propose directions for future work.

Chapter 2
Background on Virtual Memory

This chapter first introduces virtual memory technology, the segmented virtual memory, the paged virtual memory and we compare the advantages and disadvantages of paged virtual memory technology with segmented virtual memory technology. After that, we introduce the background of swapping.

2.1 Virtual Memory Management

Virtual memory is a modern memory management technique which allows programs load part of its data and code into the memory first, and then requesting other data in on demand fashion. This technique makes programmers have the illusion that they have a wide range of contiguous addressable memory space, virtual memory, which can be read/write directly. With the help of virtual memory, programmers do not have to worry about memory hierarchy. Programmers do not have to modify the programs even when the capacity of memory changes or the content of a program module changes or the network configuration changes. Virtual memory has the following advantages [17]:

- With the advent of virtual memory, programmers don't have to worry about the overlay problems which happen at when the an application needs more memory than the computing system currently has. It also helps solve the issues of partitioning and relocation which happen at when it comes with multiprogramming. Computer algorithms can be designed without considerations of the parameters of the memory configuration.

- Virtual memory brings very good protection to the operating system. A process can only access a limited number of objects which is defined in its protection

domain. The operating system will enforce a process can only reference to a memory region which is stated in its protection domain. This check could be done by hardware easily and it is very efficient.

- With the help of virtual memory, it is possible for programmers to combine separately compiled, sharable software components into their programs without worrying about arrangements other than interfaces, and without linking the software components into an address space manually.

- Virtual memory shares the same idea of object oriented programming. With virtual memory, program objects can be provided in on demand fashion. They are able to be separately designed, compiled and freely shared and resued throughout a distributed system [16, 41].

- Virtual memory is very helpful for parallel computation among machines and multicore computers. Virtual memory help reduce communication costs in distributed system. If programmers don't have a common address space, program data can only be passed through message passing. For message operations, the same data need be copied at least three times: first from the sender's process local memory to a local system buffer, second copy the data from sender's system buffer to a receiver's system buffer by the network, third copy the data from the receiver's system buffer to receiver's process local memory. With virtual memory, communication costs can be reduced by two-thirds as it only needs copy the data when the data is referenced.

The predictions that one day, main memory would be large enough to hold all the data has never come true and there is little hope and reason to believe they ever will. For scientific comptutation, it is common for them to have Tera Bytes for

computation. For personal computers, all applications's demand grow exponentially. Virtual memory is mainly designed as an extension of primary memory designed for multiple programs run concurrently. Memory segmentation and paging are two main techniques for virtual memory management. In the following sections, we will introduce about memory segmentation and paging.

2.1.1 Memory Segmentation

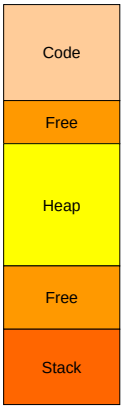

Figure 2.1: An example of program address space in segmentation.

Memory segmentation divides the virtual address spaces into several variable-length chunk of memory and the size of a memory segment can be even as small as one byte. Typically, as shown in Figure 2.1, a program address space is divided into several segments such as code segment, heap segment, stack segment. These segmentations represent a natural partition of a program, so this partition is viable to programmers [19, 43].

Each segment has the following attributes, a base address, a size and a set of permission bits. The permission bits specify which kind of access can be made by the program such as read, write, execute and etc. A virtrual address consists of a segment number and an offset. A process can only access an address within the range

of the segment specified by the virtual address and this access should be allowed by the permissions bits. Otherwise, an exception like a segmentation fault would be generated. A memory management unit (MMU) is responsible for translating a virtual address into physical address, and performens permission checks for the address to make sure the address has the permission to be assessed.

Also, a segment has a flag which indicates whether the segment is in the main memory or not. If the process wants to access a segment that is not in the main memory, an exception would be raised and the operating system would try to read the content from the secondary storage such as hard disks, solid state drives, and etc.

Memory segmentation technique has a lot of advantages. For example, segmentation is a very nice abstraction for sharing data, makes it convenient to share memory between processes with appropriate protections. Each segment can move independently and one segment move into secondary storage will not interfere with other segments. However, it has the following disadvantages. Segmentation might make serious fragmentation in the memory and makes the memory management too complicated as the size of each chunk is different. Also, as the whole segment must be present in the main memory or in the secondary storage together, the memory utilization might be very inefficient. For example, there might be only a very small fraction of data in a very large segment needs to be accessed frequently. It is a huge waste of memory to keep a large segment in the memory just in order to read only a samll fraction of it and to read the whole segment into main memory from the secondary storage in order to read such a small fraction of data is very low efficient and user experience might be seriously deteriorated. To cope with these issues, modern computer architectures usually use paging in virtual memory management.

2.1.2 Paging

Paging is a memory management technique used in modern computer operating systems by which a computer can move data between secondary storage and main memory. The smallest unit of data in paging is a page, a fixed size contiguous chunk of virtual memory addresses. It is also the smallest unit of data move between main memory and secondary storage, such as a solid state drive, a hard disk drive and etc. The size of a page depends on computer processor architecture. Typically, the size of a page in a computer system is the same, for example, 4KB. However, page sizes in a computer may have several different page sizes, such as 8KB [43, 4].

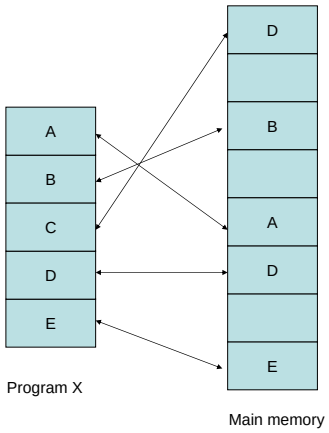

Figure 2.2: An example of program address space in paging.

Compared with memory segmentation, the most advantage of paging is that it allows the physical memory address allocated to a process not necessarily to be contiguous. For example, in Figure 2.2, a program space is devided into five different pages and these pages are mapped into main memory in pages, not necessarily contiguous. In memory segmentation, each segment needs to be physically contiguous on the physical memory. This makes memory segmentation have a serious memory fragmentation problem. Also, a fixed-size for each page makes memory management

easier compared with memory segmentation that has a variable length for each segment.

Many modern operating systems adopt paging for its memory management. As memory segmentation provides a good security protection while paging provides high memory management efficiency, some computer architecture use paging and memory segmentation together. Such a system divides the programming address space into different segments first, and then divide each segment into equal sized pages. Systems with such a design are typically paging predominant, just using memory segmentation for memory protection, such as IBM System/38, Multics [9]. In modern general purpose operating systems all adopt paging and we will just concentrate on paging in our following discussion.

2.2 Paged Virtual Memory

Page table is a data structure that is used to translate from the virtual addresses referenced by the processes to the physical addresses used by hardware. It has 2-level or 3-level. In a page table, each entry named Page Table Entry (PTE), represents a mapping between a virtual address and physical address. Each entry in the page table has a flag indicating whether the corresponding page is in the physical memory or not.

A computer system can have only one page table for the whole system or each different process has its own page table. If there is only one page table for the whole system, then different processes running at the same time should not use the same virtual address ranges, each one should use a different virtual address ranges. If each process has its own page table, then their virtual address ranges could be overlapped, as the same virtual address of different processes could be redirected to different physical addresses.

When a process try to access a virtual address, it first needs to determine the physical address and this is typically done by hardware like memory management unit. If the requested page is in the main memory, the process can simply continue execute. Otherwise, a page fault exception is raised and the operating system needs to select an empty page in the main memory to hold data reading from the secondary storage. Then operating system will load the requested data from the secondary storage to the prepared empty page and update the corresponding page table entry. After this, the operating system will return control to the process, retrying the instruction that caused the page fault.

If there is enough physical memory holding all processes data, there is no problem in selecting an empty page for loading data. However, if there are not enough pages to store all data or the empty pages is below some threshold, some pages should be selected to go to the secondary storage. If the data in the page belongs to some files on the secondary storage, and the data is different from the version on the secondary storage, for example, the data has been modified since it is read into memory or writing new data into a new file, then the data on this page needs first write onto the secondary storage before free this page. If a program wants to access a paged out page again, then a page fault would be generated to read the requested data into memory again.

2.2.1 Page Replacement

Page replacement algorithms are used to select the memory pages to paging out or swapping out to secondary storage such as hard disks, solid state drives when the number of empty memory page is less than a threshold or there is no empty memory page in the computer system. As reading/writing page from/to secondary storage needs a lot of I/O stall time and read/write performance on secondary storage is orders of magnitude slower than CPU (Central Processing Unit), if paging in/out

happens too much, user experience would be seriously deteriorated. If the page replacement algorithm is not well designed, the computer system might come into a problem named thrashing.

Thrashing, when a computer does not have enough main memory and main memory could not efficiently hold programs' working set, the computer system might come into a state which the computer system has a high rate of writing pages into swapping device and reading pages from it. This leads to very low CPU utilization and the throughput, responsiveness and latency of a computer system might degrade by multiple orders of magnitude. Hence user experience is seriously deteriorated. Even if we simply enlarge the main memory, the number of page fault generated during programs running might not necessarily reduced. This phenomenon is called Belady's anomaly [10].

Hence, an efficient page replacement algorithm must be chosen for the computer system. A simple metric for a page replacement algorithm is the less time needed waiting for paged in, the better the algorithm is. Typically, a page replacement algorithm uses a limited set of information to the pages provided the hardware to select which pages to swap out to minimize the total I/O stall time during programs running. As most of the time, it is impossible to know which pages are going to be accessed for a computer system running many different applications. Usually, a page replacement algorithm is an online algorithm and the computation and memory cost of the page replacement itself should be low.

Many computer systems adopt an approximation of the least recently used (LRU) algorithm, such as LIRS [29], 2Q [30]. Some replacement algorithms add user-level hints to improve efficiency, such as adding application controlled file caching, application informed prefetching and caching schemes. Some replacement algorithms record and utilize history information, such as LRFU which combines recency and frequency

information of a block together with LRU. Some replacement algorithms are based on the detection and adaptation of access regularities detections, different schemes will appllied according to reference patterns, such as EELRU [31].

2.2.2 Page Management in the Linux Kernel

The Linux kernel [11, 19] needs to track each page frame's current status, such as whether a page is free or not, whether a page is in the main memory or on the secondary storage right now. Also, the Linux kernel needs to determine what's the data in a page frame, such as data of a user mode process, data of kernel data structure, cached data of a software cache and buffered data for a device. All such state information is stored in a data structure called page in the Linux kernel.

The Linux kernel uses an LRU approximation algorithm, 2Q, for page management. This is not a strictly speaking LRU algorithm as strict LRU is difficult to implement and costs would be high. There two type of lists in the Linux kernel, active list and inactive list. The active list contains pages that are currently in the main memory and recently haven been referenced. The inactive list contains pages that are currently in the main memory too but are relatively not as active as pages in the active list, that is, they have been referenced in a short time period recently. These pages contain useful data for the system but might be get out of the main memory at any time. There is another list named free list, which contains empty pages. When the operating system needs to allocate an empty page for the system, it will search the free list first. When the number of empty pages in the free list is lower than a threshold, the system would try to free some pages from the pages that are containing useful data currently. The system will select some pages from the inactive list to page. If the selected page contains data but the data has the same version as the file it comes from, the content of the page would be just discarded and the page enters into free list. If the selected page contains data but different from the

content it comes from, the content of the page needs to written into the file first. If the page contains data belonging to a process and do not belong to any file on the secondary storage system, this kind of pages named anonymous page, and it must be saved for future programs running and it will written into s special device name swapping device. After this, the page would be added into the free list for future use.

When a new page is reading into the system, it will first put into the end of the inactive list. If this page is referenced by the system in a short time, this page will be moved to the tail of the active list. If a page in the active list is referenced in a short time, it will be moved to the head of the active list. If a page in the active list has not been referenced for a long time, it will be moved to the head of the inactive list. The operating system selects pages for paging out from the tail of the inactive list.

2.3 Swapping

As mentioned in last section, anonymous pages belong to no file on the secondary storage and they will be written into a special device named swapping device. More specifically, anonymous pages include pages:

- Pages that are belonging to the anonymous memory mapping part of a process, such as stack, heap.
- Pages that are belonging to the private memory mapping part of a process.
- Pages that are belonging to an IPC shared memory part.

These pages are private to processes and they can not just be discarded to read from a file later. Instead, they need to be saved into swapping devices.

The existence of swapping devices is very helpful for computer systems.

First, it enlarges the amount of memory to the computer system. Without swapping, it is impossible for a program running on a computer system with physical memory less than it needs during running. With the help of virtual memory and swapping, a large program is able to run even if just partial of its data is in the main memory.

Second, a large number of pages are only useful for a process initialization and system initialization. After that, they will never be used. Putting these kinds of pages onto the secdondary device will not compromise the system performance but also enlarge the available memory for all processes. It is better to put them on the secdondary device than keeping them in the main memory but never used in the future.

However, swapping has a major flaw. It needs to read/write from/to a secondary storage. The access time of a secondary storage is much slower than that of main memory. If a process involves a lot of swapping during its running, it will run much slower than its running in main memory and its response to users would be very poor.

2.3.1 Swap Area

The pages that are swapping out to the swapping device are saved in an area named swapping area. Swapping area can be a special disk partition or a large special file on a secondary device. Users can define several swapping area in their system, the maximum number of swapping area is confined by a macro MAX_SWAPFILES (defined in include/linux/swap.h), typically setting to 32. Spreading a large swapping space on multiple swapping devices that could run concurrently could help improve system performance when swapping is needed.

On each swap area, there are sequences of page slots which are used to store the swapped out page. Typically, the size of a page slot is the same size as the page in the main memory, 4K bytes. A page on the swapping device maps to a page from the

system. On the first page slot of a swap area, there stores some information about the swapping area persistently even if the computer system is shutdown. Those information is defined by the swap_header union (defined in include/linux/swap.h) consists of two structures, info and magic. The info structure part includes information about partition information, disk labels information, swapping algorithm chosen by the system, the last page slot that is effectively usable on the swapping device, number of bad page slots on the swapping device, padding information and so on. The magic structure part includes a character string to unambiguously indicate the swapping area and a magic string "SWAPSPACE2" at the end of the first page slot to indicate the end of the first page.

The data stored on the swapping device are only meaningful when the system is on. When the system is turned off, or the swapping device is switched off, or the process that own the data is killed, the corresponding data on the swapping device is no longer useful and can be simply discarded.

When a swap area is created, those fields in the first page slot described above will be initialized. Because it is common for a secondary device has some defective blocks, the program will search all the bad page slots to calculate bad page slots in the assigned area.

In each swap area, there is one or more swap extents. Each swap extent is described by a data structure named swap_extent. Actually, each extent is a group of adjacent pages on the disk. Thus, the swap_extent structure contains the index of the first page of the swap extent in the swap area, the number of pages in the extent, the starting sector number of disk containg the swap extent. For a swap area consists of a disk partition, only one swap extent. For a swap area consists of a regular swap file, there might be several swap extents as the file system might not create the swap area on only one contiguous area on the disk.

When more than one swap area is configured on the computer system, the system administrator has two choices. If all the swap area has the same or almost the same I/O performance, the system administrator could set all the swap areas have the same priority. In this case, the swapping out pages will be spread equally into each device cyclically without worrying about putting too much data on one swap area where might be the bottleneck in the future. Or, if the I/O performance of the swap areas has a huge difference, for example, if one swap area is on a hard disk while the other one is on the solid state drive, the I/O performance of solid state drive typically would be much better than hard disk. In this case, the system administrator could set the solid state drive has a higher priority than the hard disk has. When searching for a free page slot, the system will first search the swap area that has the highest priority. If no free page slot is found, the system will continue to search on the swap area has a lower priority. If no free page found on all the swap area, then the system will crash for outing of memory. But typically, this would not happen, as we can usually set the swap area of huge size, tens of GB.

Each swap area uses a data structure named swap_info_struct (defined in include/linux/swap.h) to describe the swap area information. This data structure includes information about the number of swap extents on this swap area, a linked lists of swap extents, descriptor of blocking device which contains this swap area, priority of this swap area, number of pages slots in this swap area and so on.

The swap area also contains a pointer which points to an array of page slots of a maximum size MAX_SWAPFILES (usually is 32768) on this swap area. There is a counter on the page slot to indicate how many processes using that page slot. If the counter on a page slot is zero, this page slot is free, available for swapping. If the counter on a page slot is large than zero and less than a macro SWAP_MAP_MAX (typically equal to 32767), this page slot has been shared by the value of counter

processes. If the value comes to SWAP_MAP_MAX, then the page slot will store the content permanently and never be changed any more. If the value of the counter on a page is SWAP_MAP_BAD, then the page is defective and never will be used.

2.3.2 Page Slot Identifier

When a page is swapping out, its corresponding address will be written into its corresponding page table entry so that the page could be found quickly when used next time. A swapped out page can be unambiguous determined by the swap area index in the swap area array and the page slot index in the swap area. As the first page of a swap area is always reserved to store some swap area configuration information, the first available page slot can be start at the first non-defective page slot in the page area which might be as small as 1.

The page slot identifier consists of the following information, page slot index and swap area index. The least significant bit of this data structure is always 0, which is also the place of the present flag in the page table entry. With this design, the operating system can always figure our whether a page is in the main memory or not in O(1) time. There three possible cases to identify the value in a page table entry:

- The value of the page entry is 0. This means the corresponding page does not belong to the process address space at all, or the corresponding page has not been assigned to the process yet.

- Least significant bit is 0 but the value of the entry is not 0. This means that the corresponding page is swapped out.

- Least significant bit is not 0. This means that the corresponding page is in the main memory.

The number of bits that is available for identifying a page slot determines the maximum size of a swap area. Usually, there is 24 bits available for identifying a page slot on the swap area on the 80*86 computer architecture. Thus, the maximum size of a swap area on a 80*86 computer is 64GB, typically large enough for personal uses.

2.3.3 Swap Cache

With the help of swap cache, many difficult synchronization problems, such as, multiple processes might try to read in the same swapped out page at the same time, a process that might read in a page that is being swapped out by the oeprating system, has been solved.

The key design is that every process must check whether the swap cache contain the affected page before they start to read in a swapped out page or swap out a page. With this design, all concurrent swap operations performed on the same page will always affect the same page frame.

For example, consider the case that two processes sharing the same swap page and all tries to read in this swapped out page at concurrently. The first process first tries to read in the swapped out page and the system will start the swap in operation. For a swap in operation, first, it will check whether the corresponding page frame is in the swap cache or not. If the page is included in the page cache, then the system will simply use the page frame descriptor and put the process into sleep until the read in data I/O operation has already been completed. If the page frame is not included in the swap cache, the kernel system will allocate a new page frame and put it into the swap cache. Then, the system will start to read in the page from the swap devices. Meanwhile, if the second process begin to issue swap in operation to read in this shared page. The system will first check swap cache too. As right now, this shared page is already included in the swap cache, the system will simply put the second process into sleep until the read in I/O operation finished.

Before a page actually is actually swapped onto the swapping device, every page needs to be first put into the swap cache. With this design, it solves the problem of multiple processes try to swap in and swap out the same page concurrently beautifully. Let us consider the case when two processes share one page frame. At first, this shared page has two owners and the reference to this page frame is put into the page table entries of both processes. When the kernel system choosees this page for swapping out, this page will be put into swap cache first. Right now, this shared page has three owners, while only the page slot in the swap cache is referenced solely by the swap cache. Then, the kernel system will try to remove the ownership of the two processes to make the swap cache as the only owner for this page. If this operation is completed, swap cache becomes the only owner of this page and the kernel system will remove the page from the swap cache and put the page frame into the free page list. Let us consider the case that while the page is being swapped out to the swapping device, one process tries to read this page. Thus, one page fault is raised. The page fault hander finds out that the desired page frame is in the swap cache and it will update its page table entry back to its physical address of the process issuing the read in operation.

Simply speaking, swap cache can be viewed as a transit center contain the page descriptors of the swapping pages that are currently swapped in or swapped out. When the swap I/O operation is completed, the page frame descriptor will be removed from the swap cache.

Chapter 3
Related Work

This chapter talks about the characteristics of hard disks and solid state drives, how Linux operating taking the characteristics of hard disks into consideration for its swapping and the existing work on making use of solid state drives for swapping.

3.1 Characteristics of Hard Disks

Hard disk has been used as the main storage medium for couples of years. It is the main storage medium for computing servers and personal computers. More than 200 companies in the world are producing hard disks, and the main manufactures are Toshiba, Seagate and Western Digital [3].

A hard disk drive, essentially consists of a spindle holding a couple of platters, is mainly used for storing and retrieving information on it. A hard disk drive could retain the data on it even when power if off. A platter is a flat circular disk made from non-magnetic materials covered with magnetic materials which record data on it. On each platter, there are thousands of concentric circles named tracks. Each track is divided into thousands of sections. A sector is the minimum data storage unit on hard disks and each sector stores 512 bytes of data. Each sector might have a different length with the same size. The density of sectors on the tracks decides the capacity of a hard disk drive. As the outer track is longer than inner track, the outer track might have much more number of sectors than inner sector has or each track has the same number of sectors which are depends on manufacturers' implementation. There are a lot of read/write heads on an actuator where read/write information from/to disk tracks. The platter spins at an extremely high speed, such as 7200 Rotations Per Minute (RPM), 15000 RPM, which has the effect of presenting the data to disk head

at a terrific speed. That's why hard disk drives have a very high efficiency under sequential reads or writes. However, for random read/write access, the disk head needs to move from one track to another track first, then wait disk head comes to the specific sector. The disk head might have to move as far as the radius of the platter in order to move to the right track. And even on the right track, the disk head might have to wait as long as half disk rotation. Thus, the efficiency of random access on hard disk drives is relatively low compared with sequential access.

Today, the price of a common hard disk is very cheap. A hard disk of one Terabytes (TB) now sold on market less than 200 dollars. Capacity is very redundant for a hard disk. Typically, a hard disk could enough for a common user. The main limit of a hard disk drive is due to its mechanical nature. It's access time is relatively slow especially compared with CPU. With Moores Law, the microprocessors improved dramatically, they become smaller, denser and more powerful. From 1970 to 2005, the functions that a CPU could perform is increased over 40% per year on average, the clock rate is increased over 30% per year on average. While the hard disk drive, with capacity increased by 1000 times from 1985 to 1995, its access speed is improved by 2 times. In the 1980, the average SRAM access time would take 0.3 CPU cycles, the average DRAM access time would take 0.37 CPU cycles. While the average hard disk seek time is 87000 CPU cycles. In 2000 year, the average SRAM access time would take 1.25 CPU cycles, the average DRAM access time would take 37.5 CPU cycles, while the average hard disk seek time is 5000000 CPU cycles. From this point of view, the hard disks in 1980 are even more than 57 times faster than their descendant in 2000.

Right now, for a typical 500 GB Seagate hard disk with a 7200 RPM, its average latency is 4.17 ms, seek time is 11ms, however, its internal I/O data transfer speed has around 150MB/s. Although the access time of hard disk is relatively slow, the hard disk has a relatively high throughput in its sequential access. If we can use

the hard disk drive in a smart way, the computer system could have a relatively good performance even if we use a device much slower than DRAM. Most of existing technologies about optimizing hard disk drives access performance are all making use of its high throughput sequential access.

3.2 Linux Swap Management for Hard Disks

When operating system comes into a situation that has a lot of swapping operations, the operating system needs to wait a page has been successfully wrtten in to the swap device before its next execution and the operating system might need to read the data from the swap devices back to the operating system when programs future demanding on it. To minimize the waiting time for the I/O completion, the Linux kernel has done a lot of special designs for hard disk drives as its swap devices based on the simple observation that the sequential access on hard disk drives are much faster than its random access.

When the Linux operating system plan to reclaim memory page, it will free a lot of memory pages in a short time, that is, the kernel will try to swap out a large number of memory pages in a short time. In order to try to minimize the I/O time to store these pages on the swapping devices and the disk seek time to find these pages when reading them back, the linux operating system tries to store all these pages in a contiguous free page slots.

There are two simple methods to store the swap out pages on the swap devices:

- Always tries to search a free page slot from the beginning of the swap area. With this strategy, the average swapping out operations time might be increased as free page slots might be scatter far away from each other. But if the swap area is not used too much, it might have a benefit of low swapping in operation time as the disk head don't have to move too far away.

- Always tries to search a free page slot from the last allocated free page slot of the swap area. With this strategy, the average swap in operations' time might increase as the page slots that the operating system want to read into memory might be scatted far away from each other. However, usually, the operating could find a contiguous free page slots easily without having to move disk head in a large range.

To take advantage of both strategies, the Linux operating system chooses to use a hybrid strategy. The Linux operating system will always try to search a free page slot from the last allocated page slot until one of the following conditions reaches:

- It comes to the end of a swap area. This strategy tries to make as many as page slots are allocated in the same swap area.

- There are SWAPFILE_CLUSTER (typically is 256) free page slots were allocated since last restart to the beginning of the swap area. This strategy tries to make pages are not scatted too far away from each other but also take advantage the high efficiency of allocating free page slots from the last allocated page slot strategy.

If there is only one swap area that has the highest priority and it has free page slots, the operating system will always try to allocate free page slots from this swap area. If there are more than one swap area with the same priority, the operating system will try to allocate free page slots in a round robin fashion from each swap area that has the same priority. If there are several swap devices which are all hard disks, the swapping could get a relative high parallel performance.

Also, when the operating system tries to swap in the swapped out pages on the swap devices, it will try to read some contiguous page slots (typically 8, including the

requested one) together. This prefetching will read up to 8 contiguous pages from the requested page, stop prefeching when meet a free or defective page slot, or already read 8 pages. This strategy takes advantage of the good sequential read performance of hard disks. If the prefetching pages will be used in near future, it saves additional swap in operations. If the prefetching pages will not be used in the future, actually, it might add more swap in and swap out operations to the system. Typically, as the swap devices contain free page slots and bad page slots, the prefetching mechanism usually could not read 8 pages together making prefetching less efficient.

3.3 Characteristics of SSD

Right now, the primary secondary storage is flash memory in the form of solid state drives. Hard disk drives are still the primary secondary storage due to its huge capacity and cheap price per unit. But solid state drives are replacing hard disk drives in areas like portable electronics where physical medium size, speed and durability are more important than capacity and price.

A solid state drive is a data storage device consists of flash memory and integrated circuit to store data permanently. There are mainly two kinds of flash memories, NOR and NAND. NOR flash memory, which was designed as a more economical and efficient rewritable ROM to store code than EPROM (Erasable Programmable Read Only Memory) and EEPROM (Electrically Erasable Programmable Read Only Memory), could support random access in bytes and write operations are relatively slow as it was expecting has much more read operation than write operation. NAND flash memory, which was designed as a more economical and efficient memory to replace current hard disk drives, has a much more dense capacity and only allows access in unit of pages. Solid state drives in the market right now are NAND flash memory based [14].

There are two types NAND flash memory, Singled Level Cell (SLC) NAND and Multi Level Cell (MLC) NAND. Each cell in a SCL flash memory could only store one bit while each cell in MLC could store two bits and more. Compared with the life expectation, SLC NAND is usually much longer than MLC NAND, while the price of SLC is much more expensive. Considering the price and capacity, most low end and middle end solid state drives are using MLC NAND to produce a high price per unit storage medium, while the high end solid state drives tends to use SLC NAND.

Solid state drives use an electronic interface to compatible with traditional block input/output (I/O) hard disk drives. For most applications, computing equipments could use it as a storage device replacing hard disk drives directly. Solid state drives do not use any moving mechanical parts, which makes it distinguish from traditional mechanical storage medium such as hard disk drives, floppy disks. Because of the mechanical moving head in the hard disk drives, the average latency and average seek time in the hard disk drives are relative large. Compared with them, solid state drives have much lower access time and latency to a block, higher sequential and random throughput, especially the random access. While the price of solid state drives are continuing decrease, the price of solid state drives are still much more expensive than hard disk drives.

In a NAND solid state drive, the flash memory consists of a number of chips, each consists of a number of planes. Each plane typically consists of thousands of blocks and some registers as a buffer. Each block typically contains a number of pages, such as 64 pages, 128 pages. Each page contains a data part typically of size 2KB or 4KB and a metadata part which contains the error check information and other information.

A flash block could not be overwriting in place. Instead, it needs first erase a flash block before the over write. Before erase a block, flash memory needs first copy

the useful data into other pages. A block typically of size between 4KB and 128KB while a page on a flash typically of size from 512B to 2KB. Thus the erase operation is a very slow operation, often an order of magnitude slower than a common write operation. Each flash memory block has a limited erase cycles. Typically, a MLC flash memory has about 10 thousand erase cycles, while a SLC flash memory has about 100 thousand erase cycles. After that, the flash block has been worn out and the cell can no longer store data reliably, the bit error would be extremely once the flash memory block erasure limit is exceeded. Usually, flash memory chip manufacturers will allocate extra flash memory to replace the wear out blocks.

In flash memory, data is written in unit as pages. However, data on the chip can only be erased at the unit of block while each block contains a large number of pages. Some of the pages in the block might no longer need and can be discarded while some of the pages in the block contain useful data and needs to be read from the pages and written into another pages before erase. This process is named garbage collection. A solid state drive will not perform garbage collection until its utilized capacity has reached some threshold. After that threshold, garbage collection will happen time and time with new data written in solid state drive. Each solid state drive employ some kind of garbage collection while they might be different between each other depends on manufacturers. Garbage collection contributes a lot of write amplification in the solid state drive.

Typically, most workloads in real world would exhibit some kind of locality in their access pattern. Some parts of the data would be access much more frequently than other parts of the data, for example, metadata information in the file system. If some blocks are programmed and erased much more frequent than other blocks, then this block would be easily wear out which made the solid state drive ending its life time early. To cope with this problem, many solid state drive manufacturers adopt

a mechanism named wear leveling, aim to make writes to each block in the flash as even as possible. In ideal case, this mechanism would make every block in the solid state drive to its maximum life time so as the flash maximum its lifetime. However, the wear leveling process itself needs to move the data which are not changing into other blocks so that those data which are changing more frequently could be written those blocks which are not using that frequently. This is serious write amplification for solid state drive and could reduce the life time of solid state drive.

Over provisioning is the capacity difference between the physical capacity of a flash memory actually it has and the logical capacity which the users are able to use. Typically, there are three levels of over provisioning in the flash memory:

- The first level of over provisioning comes from computation. Typically, an electronic storage like solid state drive uses 2^{30} bytes to represent one GB while the hard disks and solid state drive vendors uses 10^9 bytes to represent one GB. There is 7.37% $(= (2^{30} - 10^9)/10^9)$ difference between these two values which is not taken into the total over provisioning number.

- The second level of over provisioning comes from the manufacturers. Typically, there is a difference between the physical capacity and the actual available space to the users. For example, for a 128GB physical capacity solid state drive, the manufacturers might report is as a 100GB, 120GB or 128GB solid state drive.

- The third level of over provisioning comes from the users configuration. Typically, the solid state drive manufacturers allow users to set some additional spaces on solid state drive for over provisioning.

Over provisioning does take away additional spaces from users, but it helps to reduce write amplifications during garbage collection, wear-leveling which in fact increases solid state drive life endurance and helps improve performance.

3.4 Related Work on Reducing Writes to SSD

There are numerous works in the literature concerned with various aspects of solid state drives, including designs of its internal address mapping, wear leveling, and garbage collection. In this section, we briefly review the most closely related efforts in reducing write cycles in the use of solid state drive for virtual memory, and in the integration of solid state drive and hard disk for storage systems.

In the FlashVM system, solid state drive is used as dedicated swap space for page swapping for its greater cost-effectiveness than adding DRAM [39]. To reduce writes to the solid state drive, FlashVM does the following optimizations. It checks the contents of candidate swap pages and does not swap them out if they are all zeros. During swapping in, it will check whether the requested data is all zero. If the requested page is a zero page, it will allocate a zero page in applications' address space. Also, it uses a stride prefetching with the existing prefetching strategy. First using the Linux existing prefetching strategy to read pages on swap devices, then read the pages on the swap devices in a stride way. To reduce unnecessary write traffic to solid state dives, it prioritizes the younger clean pages to be reclaimed before old dirty pages in the system. During scanning pages for reclaim, the FlashVM will skip some dirty pages with a certain probability. The best skip rate of dirty pages depends on applications. For applications consist of read mostly, this strategy will save a lot of writes traffic to solid state drives. While for applications consist of write mostly, this policy will increase more page fault as younger clean pages need be evicted in the future. To cope with this issue, FlashVM detects the write rate of applications. If write rate is high, the skipping rate will be low, otherwise, the skipping rate would be high. Flash VM also merge discard on the solid state drives to amortize the erase cost.

In another work using solid state drive as swap space, SSDAlloc, data for swapping

is managed at object granularity rather than at page granularity, where objects may be much smaller then the page size [12]. Objects are defined by programmers via the *solid state drive_alloc()* API for dynamically allocating memory. Objects are all cached in a memory buffer which is managed in an LRU way that has the benefit of fast access and small cache be used for store more objects. A page buffer is used to temporarily buffer the materialized pages which are materialized in a on demand way. This page buffer is implemented via *mprotect*, managed in a FIFO (First In, First Out) way. For those page evicted from the page buffer are converted into objects. If a page evicted from the page buffer, the only cost is rematerialized the page into memory again. SSDAlloc manages objects on solid state dirves in a hash table, uses an object pool allocator with different obejct sizes, track the access patterns of the objects for exploiting temporal locality in the replacement of the objects.

Mogul et al. propose to reduce writes to non-volatile memory (NVM), such as flash used with DRAM in a hybrid memory [33]. They consider pages used by user process and file system buffering as candidates move to the NVM, and place pages with large time-to-next-write (TTNW) on the NVM. Specifically, they consider code pages of process are good candidates to be moved to NVM while stack pages and shared pages between processes are not good choices. Pages of file types are good candidates to be moved to NVM as they are good indicators of estimating time-to-next-write (ETTNW) and many of them might be read only which is a good sign for solid state drives while pages from a temporary file are bad candidates. They also using TTNW model on file names, page history and some special large applications such as databases.

Ko et al., recognizing that current OS swapping strategies are designed for hard disk and can cause excessive block copies and erasures on solid state drive if it is used as swap space [28]. They put swapped out pages in a log structure, when a

new write added at the end of log, an old page in the log would be invalidated. This could help fully make use of the fixed I/O bandwidth and prevents the read and write operations interfere with each other on solid state drive. A garbage collector will collect the obsolete data left in the log in the background when not much I/O operations on the solid state drives. During swapping in operations, they will read pages in a block-aligned manner in order to reduce garbage collection cost as garbage collection cost would be large if just partial of the block is invalidated and also might slow down the garbage collection process.

Because solid state drive is still used as a swap device in addition to hard disk in our proposed HybridSwap system, the optimizations of the use of solid state drive in the above works are complementary or supplemental to our effort to improve solid state drive lifetime. In some of the works the Linux virtual memory prefetching strategy is tuned to match solid state drive's characteristics to improve swapping efficiency, such as allowing non-sequential prefetching and realigning prefetching scope with solid state drive block boundaries [39, 28]. In contrast, HybridSwap coordinates prefetching over the solid state drive and hard disk to hide disk access latency.

As an accelerator for hard disk, solid state drive has been used buffer cache between main memory and the hard disk and exploits workloads' locality for data caching [5, 37]. In SieveStore, they make extensive experiments reveals that although highly skew property across all servers, 1% of the most accessed blocks account for a huge fraction of total accesses. Then, SieveStore tries to find who are the most popular 1% blocks and put them in a solid state drive cache to make an cost effective storage system.

Another use of hard disk and solid state drives is forming a hybrid storage device with hard disks and solid state drives in parallel such that frequently accessed data is stored on the solid state drive [15, 36]. In Combo Drive, they use one static optimizer

to move files according to their file types instead of their actual file access, always executable files and program libraries who has a expectation of frequent access to the solid state drives while put other types file to the hard disk drives. Also, they use one dynamic optimizer to move files according to their access pattern, moving files consist of only random access to the solid state drives while putting other access pattern files to the hard disk drives. In Hystor, they identify the blocks that may result in long latencies and semantically critical blocks, put them in solid state drives to have a fast accesses performance in the future by monitoring I/O access online using frequency/request size as the metric.

A major effort in these works using solid state drives as hard disks buffer cache for optimized performance and improved solid state drive lifetime is in dynamic identification of randomly-read blocks and caching them on, or migrating them to, the solid state drive. In principle, HybridSwap has a similar goal of directing sequential page access to the disk. However, unlike accessing file data in a storage system, HybridSwap manages the swapping of virtual memory pages and has different opportunities and challenges. First, the placement of swapped-out pages on the swap space is determined by the swapping system rather than the file system. For continuously swapped-out pages HybridSwap can usually manage to sequentially write to the disk. There is an opportunity to improve the efficiency of reading from the disk in resolving page faults by placing the pages on the disk in an order consistent with their anticipated future read (or page fault) sequence. To this end, HybridSwap predicts future read sequences at the time of swapping out pages. Second, because data in a file system is structured, information is available to assist in the prediction of access patterns; for example, metadata and small files are more likely to be randomly accessed. Virtual memory pages lack such information and their access patterns can be expensive to detect. Third, swap space can be more frequently accessed than files

because it is mapped into process address space, and data migration strategies used in works concerned with hybrid disks adapting to changing access patterns are usually not efficient in this scenario. To meet these challenges, HybridSwap incorporates new and effective methods for tracking access patterns and the laying out of swap pages in the swap space.

In the domain of file I/O, there are two recent works in which the hard disk is explicitly used to reduce writes to the solid state drive. To allow disk to be a write cache for access to the solid state drive, Soundararajan et al. analyzed file I/O traces in the desktop and server environments and found that there is significant write locality—most writes are on a small percentage of file blocks and writes to a block are concentrated in a short time window. Based on this observation they cache these blocks on the disk during its write period and migrate them to the solid state drive when the blocks start to be read [22]. In contrast, swap pages do not have such locality because a page's access immediately following a swap-out, if ever, must be a read (swapping in). Therefore HybridSwap must take into account the efficiency of reading pages from the disk. The second work, I-CASH, makes the assumption that writes to a file block usually do not significantly change its contents, i.e., that the difference in content (the delta) is usually small [38]. The soild state drives will store most changed seldom blocks and mostly serve for reads, while the hard disks store the deltas between the currently accessed data and their corresponding copy on the solid state drives. To have a high read efficiency, they use a high speed compression method to compress the deltas and put them on the hard disks in a sequential way to make use the high I/O performance of hard disks. However, for virtual memory access, there is not sufficient evidence to support the assumption that the deltas are consistently small. In addition, the on-line computation required for producing the deltas and recovering the original pages with deltas could heavily burden the CPU. In contrast,

HybridSwap achieves high read efficiency from the disk by forming sequential read patterns to exploit high disk throughput.

To provision QoS assurance for programs running in virtual memory, the common practice is to prevent the memory regions of performance-sensitive programs from swapping, either enforced by the kernel [13] or facilitated with application-level paging techniques [25]. In the HPC environment swapping (or virtual memory) is often disabled (or not supported) to ensure predictable and efficient execution of parallel programs on a large cluster [42]. While HybridSwap advocates the use of the hard disk along with solid state drive as a swapping device, its integrated QoS assurance mechanism is used as a safety mechanism to prevent excessive performance loss in the effort to reduce wear on flash.

Chapter 4
Design and Implementation

When a combination of solid state drive and hard disk is used to host swap space as part of virtual memory, the performance goal is to efficiently resolve page faults, i.e. to read pages from the swap space quickly. While HybridSwap is proposed to achieve the reduction of writes to solid state drive, an equally important goal is high efficiency for reading swapped pages. To achieve these goals, HybridSwap is designed to carefully select appropriate pages to be swapped to the disk and to schedule prefetching of swapped-out pages back into memory. To this end, we need to integrate spatial locality with the traditional consideration of temporal locality in the selection of pages for swapping, evaluate access spatial locality, and schedule the swapping of pages in and out.

4.1 Integration of Locality

Because swap space is on a device slower than DRAM, the temporal locality in page access is exploited by retaining the most frequently accessed pages in memory. For this purpose, the operating system buffers physical pages that have been mapped to the virtual memory in the system page cache and tracks their access history. A replacement policy is used to select pages with the weakest temporal locality as targets for swapping out. Consecutively identified target pages are swapped together and are highly likely to be contiguously written in a region of the swap space. However, there is no assurance that these pages will be swapped in together in the future. Furthermore, the replacement policy may not identify pages that were contiguously swapped in as candidates for contiguously swapping out with sequential writes. Ignoring spatial locality can increase swap-in latency and page-fault penalty, especially

when the swap space is on disk. In the context of HybridSwap, spatial locality refers to the phenomenon that contiguous pages on the swap space are the targets of page faults occurring together and can be swapped in together. While swap-in efficiency is critical for the effectiveness of the hard disk as a swapping medium, spatial locality must be integrated with temporal locality when HybridSwap swaps pages out to the hard disk. To this end, among pages of weak temporal locality, we identify candidate pages with potentially strong spatial locality and evaluate their locality according to their access history. Only sequences of pages with weak temporal locality and strong spatial locality will be swapped to the hard disk, and those of weak spatial locality will be swapped to the solid state drive.

In the LRU (least recently used) replacement algorithm, one of the more commonly used replacement algorithms in operating systems, it is relatively easy to recognize sequences of candidate pages. HybridSwap is prototyped in the Linux 2.6 kernel that adopts an LRU variant similar to 2Q replacement [27]. Here the kernel pages are grouped into two LRU lists, the active list and the inactive list. As their names suggest the active list is used to store recently or frequently accessed pages, and the inactive list is used to store pages that have not been accessed for some time. A faulted-in page is placed at the tail of the active list, and pages at the tail of the inactive list are considered to have weak temporal locality and are candidates for replacement. A page is promoted from the inactive list into the active list if it is accessed, and demoted to the inactive list if it is not accessed for some time. Pages that have been accessed together when they are added into the inactive list will stay close in the lists. However, a sequence of pages in the lists may belong to different processes, as a page fault of one process leads to scheduling of another process and consecutive page faults are more likely to originate from different processes. Such a sequence is unlikely to repeat itself because the involved processes usually do not

coordinate their relative progress. Therefore, HybridSwap groups pages at the tail of the inactive list according to their process IDs, and then evaluates their spatial locality within each process, as shown Figure 4.1.

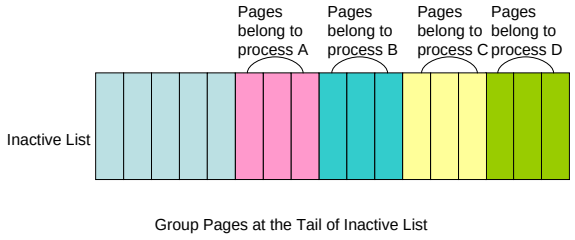

Figure 4.1: An example of grouping pages at the tail of inactive list.

4.2 Evaluation of Spatial Locality of Page Sequences

For a sequence of pages at the tail of the inactive list, we need to predict the probability of them having page faults and being swapped in together if they are swapped out. As the basis of this prediction, we check the page access history to determine whether the same access sequence has appeared before. The challenge is how to efficiently detect and record page accesses. One option is to use *mprotect()* to protect the pages of each process to detect page access when an *mprotect*-triggered page fault occurs. This option can be overly expensive because the system may not always be frequently page swapping, and not all pages are constantly involved in the swapping. Instead, HybridSwap only records a page access when a page fault occurs on the page. In this way, there is almost zero time cost for detecting page accesses, and the space overhead for recording access is proportional to number of faulted pages. Assuming that programs have relatively stable access patterns and that the

replacement policy can consistently identify pages of weak locality for swapping, the access history recorded in this manner is sufficient to evaluate the spatial locality of the sequence of pages to be swapped out.

When a sequence of pages is swapped out together and sequentially written to the disk they will be swapped in together—or sequentially prefetched into the memory—in response to a fault on any page in the sequence. However, sequential disk access does not necessarily indicate efficient swapping, because for efficiency the prefetched pages must be used to resolve future page faults before being evicted from memory. In other words, the spatial locality for a sequence of prefetched pages is characterized by how close in time they are used to resolve page faults. Quantitatively, the locality is measured by the time gap between any two fault occurrences on the pages in the sequence. Ideally, it is not larger than the lifetime of a swapped-in page, or the time period from its swap-in to its subsequent swap-out.

In evaluating locality, there are three goals in effectively recording page access. First, the space overhead should be small. Second, the recorded history must allow an algorithm to determine whether a page sequence has appeared before in the $O(1)$ time. Third, the history should be able to predict whether future page faults on the pages in the sequence would occur *together*. To achieve these goals, for each process we build a table, its *access table*, that is the same as the process's page table, except that (1) only pages that have had faults are in the table; and (2) the leaf node of the tree-like table, equivalent to the PTE (page table entry) in the Linux page table, is used to store the times when page faults occur on its corresponding page. We set a global clock that is incremented whenever a page fault occurs in the system. For the page associated with the fault we record the current clock time in the page's access table entry. We record up to two such times associated with the page's two most recent page faults. By using page fault count for the timing, rather than using

real system time, we can assess the space demand on the system buffer as each page fault requires an additional page space. If there are two page faults separated by a time period with few or no page faults in between, they are considered to be close in our timing metric as they would probably be evicted together without separated in the LRU list by other faulted pages. In contrast, the wall-clock time can not serve this purpose in this context. We also record a page's most recent swap-in time to obtain the page's most recent in-memory lifetime. When a page is swapped out, we compute the difference between the current clock time and its swap-in time to obtain the page's most recent lifetime. We calculate a moving average of the lifetimes of any swapped pages for the system and use it as a threshold to evaluate a page sequence's spatial locality. The average L_k is updated after serving the kth request by

$$L_k = (1 - \alpha) * L_{k-1} + \alpha * \mathit{Lifetime}.$$

Here *Lifetime* is the lifetime calculated for the most recently swapped out page, and α is between 0 and 1 and is used to control how quickly history information decays. In our reported experiments $\alpha = 2/3$ so that more recent lifetimes are better represented. Other experiments show that system performance is not sensitive to this parameter over a large range. In the prototype, we use 32 bits to represent a time. When the clock reaches to its maximum value it is reset to zero. (The temporary disruption that could be caused by this reset would only occur if a process swapped 16TB of data using 4KB pages—if this were an issue a 64 bit value could be used.) The space overhead for storing timestamps is then 32 bits/4KB, or 1B/1KB, a very modest 0.1% of the virtual memory involved in page faults.

Only sequences of high spatial locality are eligible to be swapped to the hard disk. When there is a candidate sequence of pages selected from the tail of the inactive

list that belong to the same process, we use the process's access table to determine whether the sequence has sufficiently high spatial locality. We first retrieve the pages' access times from the table. For each access time of a page, we calculate the difference between it and the access times of other pages. If there exists another page whose access times differ by more than the system's current average lifetime, or there are anonymous pages in the sequence that do not have history access times, the sequence's spatial locality is deemed to be low.

4.3 Scheduling Page Swapping

There are three steps for swapping out pages in HybridSwap: selecting a candidate sequence, evaluating its spatial locality, and determining swapping destinations. For each swap, we select a process and remove all of its pages from the last N pages at the tail of the inactive list, where N is a parameter representing the tradeoff between temporal locality and spatial locality. A smaller N will ensure that only truly least recently used pages are swapped but provides less opportunity for producing long page sequences and may results in more page faults in near future as memory is not enough. In contrast, overly large N can benefit disk efficiency but may lead to the replacement of recently used pages and results in more page faults as hot pages has been swapped out. Fortunately, because the list can be relatively long with today's memory sizes, N can be sufficiently large for high disk efficiency without compromising the system's temporal locality. In current Linux kernels, eight pages are replaced in each swap. Previous research has suggested that disk access latency can be well amortized with requests of 256KB or larger [40], so N is set to $p * 128KB/s$, where p is the number of processes with pages in the tail area of the inactive list and s is the page size (4KB). Sequence selection is rotated among the processes for fair use of memory.

Spatial locality is evaluated for each selected candidate sequence. If the result indicates weak locality the sequence is swapped to solid state drive. Otherwise, in

principle the sequence will be sent to the disk. A sequence that is written to the disk is recorded in the corresponding process's access table using a linked list embedded in the leaf nodes of the table recording their virtual addresses. When a fault occurs on a page in the sequence the first page of the sequence will be read in synchronously, then the other pages of the sequence will be asynchronously prefetched from the disk into memory.

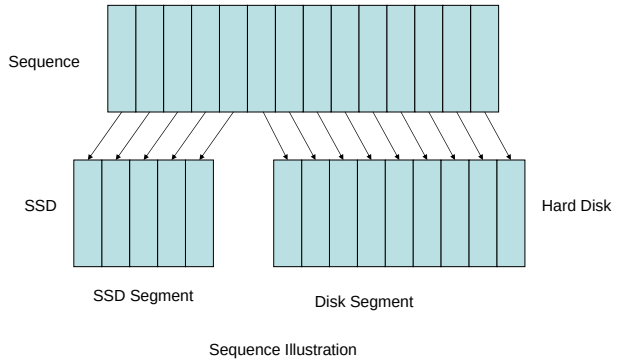

Figure 4.2: Illustration of SSD Segment and Disk Segment.

While asynchronous prefetching of pages is expected to allow their page faults to be resolved in memory, one deficiency in the swapping-in operation is the long access latency experienced by the read of the first page. To hide this latency, for a sequence intended for swapping to the disk, we examine its pages' history access times recorded in the access table to see whether they had been accessed in a consistent order. If so, we order them in a list of ascending access times and divide them into two segments. The first segment, called the SSD segment, will be swapped out to the solid state drive, and the second one containing pages of longer access times, called the disk segment, will be swapped out to the disk, as shown in Figure 4.2. If there is a fault on the page in the solid state drive segment, both the solid state drive and disk segments

are immediately prefetched. The objective of this scheduling is to hide the long disk access latency behind the time for the process to do computation on the data supplied by the solid state drive. To determine the length of the solid state drive segment, we need to compare the data consumption rate of the process to the disk access latency. To achieve this, we track the time periods between any two intermediate page faults for each process, and calculate their moving average T_{compt} with a formula similar to the one used for calculating system average lifetime. We also track the access latencies associated with each disk swap-in, and calculate their moving average $T_{disk-latency}$, also with a similar formula. If $T_{disk-latency}/T_{compt}$ is smaller than the sequence size then it is the solid state drive segment size. Otherwise, the entire sequence will be swapped to the solid state drive. In this way overly-short sequences will be swapped to the solid state drive even if they have strong spatial locality. In HybridSwap we do not need to explicitly evaluate the accuracy of the prefetching: inaccurate prefetching is caused by changing access pattern, and inconsistent access patterns are recorded in the access table, which will automatically cancel disk swapping of the involved pages.

4.4 Building QoS Assurance into HybridSwap

HybridSwap is designed to carefully select pages for swapping to the disk to attain a good balance between reducing solid state drive writes and retaining the performance advantage of solid state drive. At the same time, HybridSwap allows a user to specify the prioritization of these two goals to influence how the tradeoff is made for specific programs. We use the ratio of program stall time due to page faults and its run time as the input. This ratio is a mandatory upper bound on the cost of swapping on a program's run time. To implement this requirement, HybridSwap needs to know the current ratio, which is the ratio of current total stall time and the elapsed time of the program's execution.

If no QoS requirement is specified for a program's execution, its page swapping will be managed in the default manner as previously described. Otherwise, HybridSwap tracks the ratio of the current ratio and the required ratio. If this ratio, which we call the shift ratio, is larger than 1 we need to shift subsequent sequences towards the solid state drive when pages are swapped out. Initially a sequence's solid state drive segment is obtained as described in Section 4.3. If the shift ratio is larger than 1, the solid state drive segment size is increased by this ratio (up to the entire sequence size). If the shift ratio is still larger than 1 with almost all recent pages swapped to the solid state drive, the sequence is shifted towards memory in a similar manner. For pages to be kept in memory, we simply skip those pages when selecting pages for swapping out. If the shift ratio is consistently smaller than 1 by a certain margin, HybridSwap will shift sequences towards solid state drive or the disk. For effectively determining the placement of swapped-out sequences according to the required ratio, HybridSwap initially places the pages on the solid state drive when a program starts to run.

Chapter 5
Evaluation and Analysis

HybridSwap is implemented primarily in the Linux kernel (version 2.6.35.7) and supports a user-level QoS tool. Most code modifications are in the memory management system, for example instrumentation in the *handle_pte_fault()* function to record both major and minor page faults in the access tables, and in the *shrink_page_list()* function to select sequences for spatial-locality evaluation and swapping out. At the user level, we wrote a script to read user-specified QoS requirement from the user's command line. This tool is only needed for users requesting specific bounds on the overhead due to page swapping during program execution. In total about 1200 lines of code were modified or added.

In this section we experimentally answer the following questions about HybridSwap.

- Can the number of I/O writes to SSD be significantly reduced by using the hard disk?
- Can HybridSwap achieve performance comparable to SSD alone?
- Is HybridSwap effective for workloads with mixed memory access patterns or for very heavy workloads?
- What is the overhead of HybridSwap as implemented in the Linux kernel?

5.1 Experimental Setup

We conducted experiments on a Dell Poweredge server with an Intel 2.4GHz dual-core processor with 4GB DRAM. Except where otherwise stated, we limited the memory available for running processes to 1GB to intensify the swapping activities

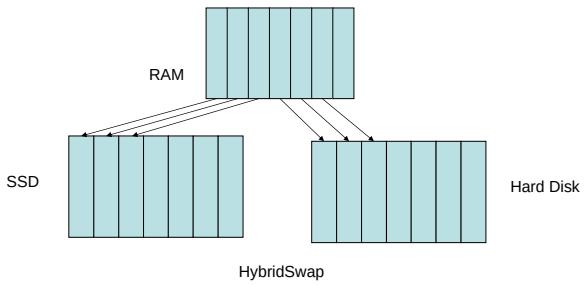

Figure 5.1: An illustration of HybridSwap configuration.

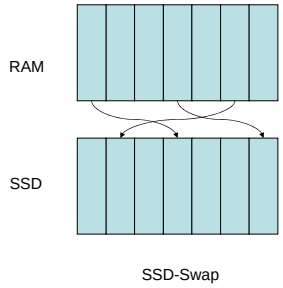

Figure 5.2: An illustration of SSD-Swap configuration.

and reveal performance difference between different swapping strategies. The server is configured with a 160GB hard disk (WDC WD1602 ABKS). We used two types of SSD devices in the experiments. One is Intel SSDSA2M080G2GC, referred to as Intel SSD hereafter, and the other one is OCZ-ONYX, referred to as OCZ SSD hereafter. Their performance characteristics are summarized in Table 5.1. Except where explicitly stated the OCZ SSD is the one used in the experiments. Native Command Queuing (NCQ) is enabled on the disk and SSDs. Per standard practice we used CFQ [2]

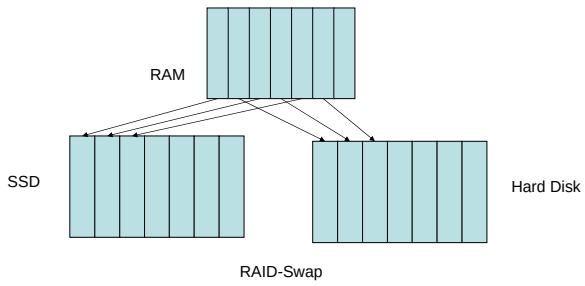

Figure 5.3: An illustration of RAID-Swap configuration.

(Completely Fair Queuing) which optimize the order of the I/O requests it receives as the disk scheduler and NOOP [7] which keeps the order of I/O requests as the order it receives as the SSD scheduler. In the experiments, we compared HybridSwap (as shown in Figure 5.1) with the swapping configuration using only an SSD, referred to as *SSD-Swap* (as shown in Figure 5.2). Linux supports swapping on multiple storage devices with a pre-configured swapping traffic distribution. To compare HybridSwap with a less-optimized hybrid swapping system, we use an SSD and a hard disk with 1:1 distribution as a Linux-managed swapping space, referred to as *RAID-Swap* (as shown in Figure 5.3). In RAID-Swap, the swap space is equally striped on the SSD and the disk, leading to a data layout similar to RAID 0.

5.2 Benchmarks

We used four real-world memory-intensive benchmarks with differing memory access patterns to form the workloads: Memcached, ImageMagick, matrix inverse, and correlation computation.

Memcached provides a high-performance distributed caching infrastructure to form an in-memory key-value store [8]. We set up a Memcached client to issue re-

	Intel SSD	OCZ SSD	Hard Disk
Capacity (GB)	80	32	160
Sequential Read (MB/s)	238	140	85
Sequential Write (MB/s)	125	110	70
Random Read (MB/s)	25	34	4
Random Write (MB/s)	10	4	0.8

Table 5.1: Capacities and sequential/random read/write throughput of the two SSDs and the hard disk. The measurements are for 4KB requests. Random access throughput indicates access latency, which is relatively very large for the hard disk. In contrast, the hard disk's sequential throughput is not much lower than the SSDs'.

quests for storing (*put*) and retrieving (*get*) data items at the Memcached server with different dispatch rates and key distributions.

ImageMagick (*Image* for short) is a software package providing command-line functionality for image editing [6]. In the experiments we enlarge a file of 17MB by 200%, and at the same time we convert the file from its original JPG format to PNG format.

Matrix Inverse (*Matrix* for short) is a scientific computation program from ALGLIB, an open-source and cross-platform numerical analysis and data processing library [1]. The implementation of matrix inverse in ALGLIB employs several optimizations, such as efficient use of CPU cache, to achieve maximal performance. The input matrix size is 4096*4096 in the experiments.

Correlation Computation (*CC* for short) is also from ALGLIB. It finds the statistical dependence between two matrices by calculating their correlation coefficient. The input matrix size is 4096*4096 in the experiments.

5.3 Reduction of SSD Writes

A major goal of HybridSwap is to select appropriate pages for swapping to the hard disk to reduce writes to the SSD and so improve its lifetime. We measured the number of page writes for each of the four benchmarks. For each benchmark we ran

	Memcached (5)	Image (2)	CC (6)	Matrix (7)
SSD-Swap	269,618	450,352	911,148	471,201
HybridSwap	170,316	273,477	712,660	396,291
Reduction Ratio	37%	40%	22%	16%

Table 5.2: Number of writes to the SSD with and without use of the hard disk managed by HybridSwap. The number of concurrent instances is given in parentheses.

multiple concurrent instances to increase the aggregate memory demand and create more complicated dynamic memory access patterns. Table 5.2 lists the total number of page writes on the SSD for each of the four benchmarks when only SSD is used and when both SSD and hard disk are used and managed by HybridSwap.

For Memcached we ran five concurrent instances. To simulate the use of the caching service by applications running on the client, we assume each instance has its own set of data items to constantly *put* and *get* in a certain time period before it operates on another set of items. The size of an item is uniformly distributed between 64KB and 256KB. The total item size is 1.95GB, for 0.95GB data on the swap space and the rest in the 1GB main memory. Memcached is pre-populated with the *put* operations. After that the client simultaneously issues queries to different data sets in different Memcached instances, with each set having 500 queries at a time. HybridSwap detects relatively strong spatial locality in the memory accesses when the same set of data items is accessed together from time to time. It reduces writes to the SSD by 37% by swapping some sets of data items to the disk. Because only sets of data items of strong locality are placed on the disk, and the sequential disk bandwidth is comparable to the non-sequential bandwidth of the SSD, we achieved only 0.8% slowdown in terms of the number of packets (or queries) transferred between Memcached clients and server per second as shown in Table 5.3.

We ran two concurrent instances of Image, which has a sequential memory access

	Memcached (packets/s)	Image (s)	CC (s)	Matrix (s)
SSD-Swap	16,734	108	431	864
HybridSwap	16,597	103	400	860
Improv. Ratio	-0.8%	4.6%	7.1%	0.5%

Table 5.3: Performance of the benchmarks when either SSD-Swap or HybridSwap is used. Memcached's performance is in terms of throughput (packets/s), and the other benchmarks use run time (s) as the performance metric.

pattern and requires 2.4GB memory. An interesting observation about the benchmark is that not only is the number of I/O writes reduced by 40%, but the run time of the benchmark is also reduced by 4.6% (Table 5.3). This is because aggressive prefetching enabled by HybridSwap based on sequences on the hard disk reduces the total number of major faults from 53,557 to 37,929 (29% reduction), and transforms the remaining faults to minor faults (hits on the prefetched pages). Another reason is that the SSD and hard disk can concurrently serve requests with higher aggregate I/O bandwidth than SSD alone.

We ran six instances of CC, which requires 3.4GB memory. The results show that 22% of writes to SSD are eliminated by HybridSwap due to its detection of access sequences with strong locality and their efficient swapping to disk. However, we found that compared to the Image benchmark there is a larger percentage of writes to the SSD. To gain an understanding of this we collected the sequences generated during the executions of the two benchmarks. Figure 5.4 shows the cumulative distributions of their sequence sizes. We observe that for CC and Image, 87% and 75% of the sequences have fewer than 30 pages, respectively, that is, CC has more shorter sequences. With short sequences, even if they are qualified to be swapped to the disk, pages in their SSD segments can be a large proportion of the sequences and are stored on the SSD. With more pages on the SSD the relative performance of CC

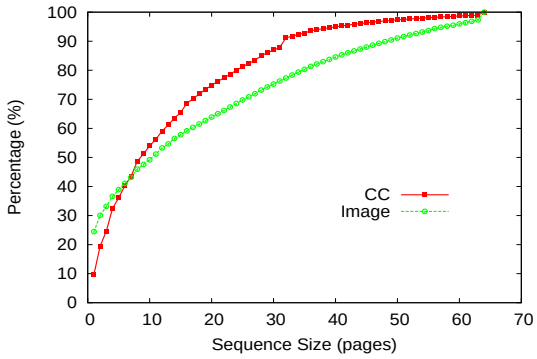

Figure 5.4: Cumulative distributions of sequence sizes for the Image and CC benchmarks.

is greater than that of Image (Table 5.3).

Because the Matrix benchmark has small memory demand (260MB) we ran seven instances to stress the swap devices. Although it produces a significant number of short sequences, HybridSwap can still exploit the detected locality to carry out disk-based swapping without compromising performance. Writes to SSD are reduced by 16%. Our measurements show that the swapped-out data is distributed on the SSD and on the hard disk at a ratio of 5:1 when the swap space reaches its maximal size.

In the experiments presented in the next several sections we select only one or a subset of the benchmarks that is most appropriate for revealing specific aspects of HybridSwap.

5.4 Effectiveness of Sequence-based Prefetching

HybridSwap records access history to identify sequences of strong locality for swapping to disk and to enable effective prefetching afterward. Lacking information with which to predict page faults, Linux conservatively sets a limit of eight pages for each prefetch. In contrast, HybridSwap can prefetch as many as 64 pages in one swapping-in. In this section we compare the number of major page faults associated

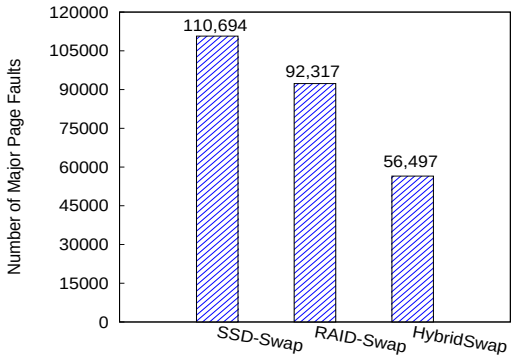

Figure 5.5: Number of major faults when running the CC benchmark with SSD-Swap, RAID-Swap, and HybridSwap. For SSD-Swap and RAID-Swap we use the Linux default read-ahead policy.

with HybridSwap, SSD-Swap, and RAID-Swap when running CC. As shown in Figure 5.5, HybridSwap reduces page faults by 49% and 39% compared to SSD-Swap and RAID-Swap, respectively. By exploiting consistent page access patterns such as row-based, column-based, and diagonal-based access, HybridSwap can detect a large number of long sequences of strong spatial locality. There are two factors contributing to HybridSwap's small major-page-fault count. One is the high I/O efficiency in its page swap-in, which helps make pages available in memory before they are requested to resolve faults. The other is the accuracy of its prefetching, which makes the swapped-in pages become the targets of minor page faults. Using only one device, SSD-Swap has the most major faults.

5.5 Effect of Memory Size

Memory size is inversely correlated to the amount of data swapped out of the memory, so we can vary the memory size to influence the swapping intensity and correspondingly affect the relative performance advantages or disadvantages of HybridSwap. In the following experiments we ran Memcached with different amounts of

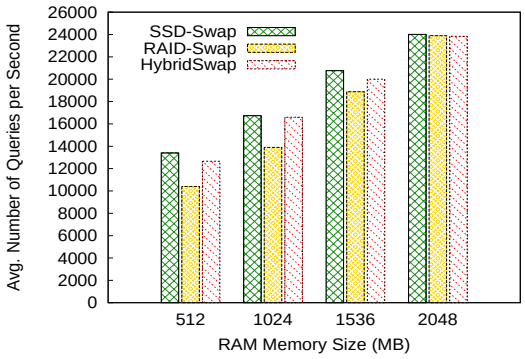

Figure 5.6: Memcached throughput in terms of average number of queries served per second with different memory sizes and different swapping schemes.

memory to determine whether the program's relative throughput is substantially affected by a swapping-intensive workload. We increase the memory size from 512MB, 1024MB, 1536MB, to 2048MB. As shown in Figure 5.6, when memory size is small the throughput of Memcached in terms of average number of queries served per second is higher for SSD-Swap than that for HybridSwap. However, the loss in throughput is small—only 5.5%—and is accompanied by a 40% reduction of writes to SSD. Compared to RAID-Swap, HybridSwap's throughput is higher by 22% because Linux does not form sequences that enable effective prefetching. As the memory size increases more accesses can be served in memory, resulting in correspondingly increasing throughput. When the memory size is 2GB, 98% of the programs' working set is in memory and there is little difference between the throughputs of the three schemes.

5.6 Insights into HybridSwap Performance

In this section we investigate how the performance of HybridSwap relative to that of SSD-Swap changes with differing types of SSD devices, and why the performance of HybridSwap is much higher than that of RAID-Swap even though both use the same combination of SSD and hard disk. We ran Image and measured its run time

	SSD-Swap	RAID-Swap	HybridSwap
OCZ	108/53557	144/64160	103/37929
Intel	104/58459	143/62457	104/35218

Table 5.4: Run time/number of major faults during the executions of the Image benchmark with different swapping schemes and different types of SSDs.

and total number of writes to SSD for the two types of SSDs using each of the three swapping schemes, as shown in Table 5.4. Table 5.1 shows that the difference in throughput of the two SSD devices ranges from approximately 15% to 150%, and generally the Intel SSD has higher performance. However, the relative performance shown in both program run time and number of major page faults is minimally affected by the SSD types. The reason is that in HybridSwap, SSD is mainly used for random access and the hard disk is used to serve sequential access. While a significant portion of requests are identified as sequential and served on the disk, the role of SSD on HybridSwap's performance is important. We expect that an SSD with much higher performance would place HybridSwap at a disadvantageous position in terms of program run time. However, such an SSD would be much more expensive so the need to extend its lifetime by using HybridSwap could be higher.

Table 5.4 also shows that HybridSwap consistently and significantly outperforms RAID-Swap by reducing both writes to SSD and run time of the benchmark for both types of SSD devices. Apparently this performance advantage comes from the different usages of the hard disk. Figure 5.7(a) and Figure 5.7(b) show access addresses on the hard disk in terms of disk LBN (Logic Block Number) when HybridSwap or RAID-Swap is used, respectively, in a sampled execution period of 0.2 seconds. The lines connecting consecutively accessed blocks are indicative of disk head movement. When RAID-Swap is used the disk head frequently moves in a disk area whose LBNs range from 30 to 700,000 indicating low I/O efficiency and high page-fault service

time. When Hybrid-Swap is used pages in the same sequence are written and read sequentially. This shows that HybridSwap's performance, which is comparable to that of SSD-Swap and is much higher than that of RAID-Swap, should be mainly attributed to its improved disk efficiency.

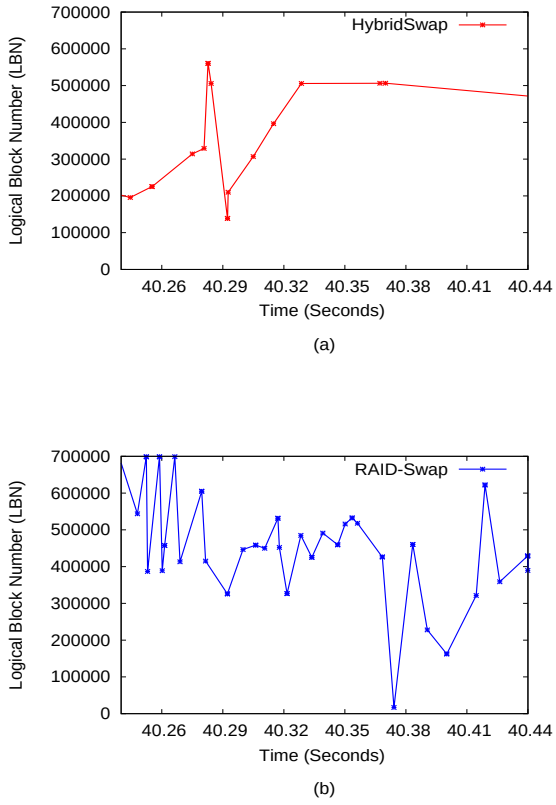

Figure 5.7: Disk addresses, in terms LBNs of data access on the hard disk in a sampled execution period with HybridSwap (a) and with RAID-Swap (b).

5.7 Multiple-program Concurrency

Concurrently running multiple programs could potentially lead to swapping of a large number of random pages and compromised disk efficiency. In this section we use the Matrix benchmark to evaluate the performance of HybridSwap with varying concurrency from 4, 5, 6, to 7 program instances.

With four concurrent instances there are only 0.2GB of data to be swapped out and the I/O time spent on faults accounts for only 1.7% of total execution time. As the concurrency increases the total number of page faults significantly increases by 32 times (Figure 5.8(a)), and I/O time for swapping increases by 4.5 times (Figure 5.8(b)). From the figures we make three observations: (1) When the system has a swapping-intensive workload, RAID-Swap should not be used as it produces many more major page faults and spends much more I/O time on swapping; (2) SSD-Swap cannot sustain performance for workloads with highly intensive swapping because the bandwidth of one SSD can be overwhelmed; and, (3) HybridSwap retains its advantage in the small number of major faults and low swapping time even with very high concurrency. This suggests that a heavy workload does not prevent HybridSwap from forming sequences and efficiently using the disk for swapping.

When HybridSwap is used 49% of page faults are eliminated by effective swap-page prefetching based on faults history. Because pages are read and written sequentially on the disk and from the two devices in parallel, HybridSwap achieves even better performance than SSD-Swap. Total swap I/O time is reduced by 20% and 43% compared to SSD-Swap and RAID-Swap, respectively.

5.8 Bounding the Page Fault Penalty

While swapping data to the storage devices, especially hard disk, can severely extend a program's run time, HybridSwap provides a means to bound the page fault penalty on run time. To demonstrate this QoS control we run three instances of Image. Figure 5.9(a) shows that the ratio of the aggregate page fault penalty and the run time, as a function of elapsed run time, without any QoS requirement. Because no instance has a bound on the ratio, all three make best effort in the use of virtual memory and their respective ratio's curves are nearly identical. Initially the ratios rise as each builds up its respective working set and page faults increase as more of their

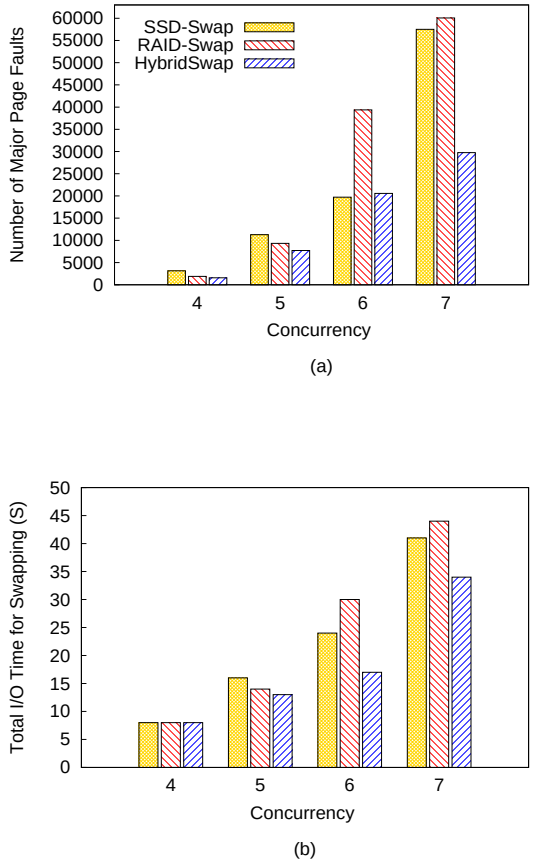

Figure 5.8: Major faults (a) and total I/O time (b) spent on the swapping in the running of the Matrix benchmark with varying degrees of concurrency.

working sets are swapped. In the curves there are segments that are not continuous because we only sample the ratio at each swap-out to reduce the overhead. We next attempt to set a bound on the ratio for one or more of the instances to prioritize their performance. Because the memory size is limited, which is smaller than one instance's working set, an instance prioritized with a tighter bound would move some or all of the working set that would be placed on the hard disk without a QoS requirement to the SSD. Figure 5.9(b) shows the ratios of the instances when we set a bound of 70% on the ratio of Image instance 1. By enforcing the QoS requirement

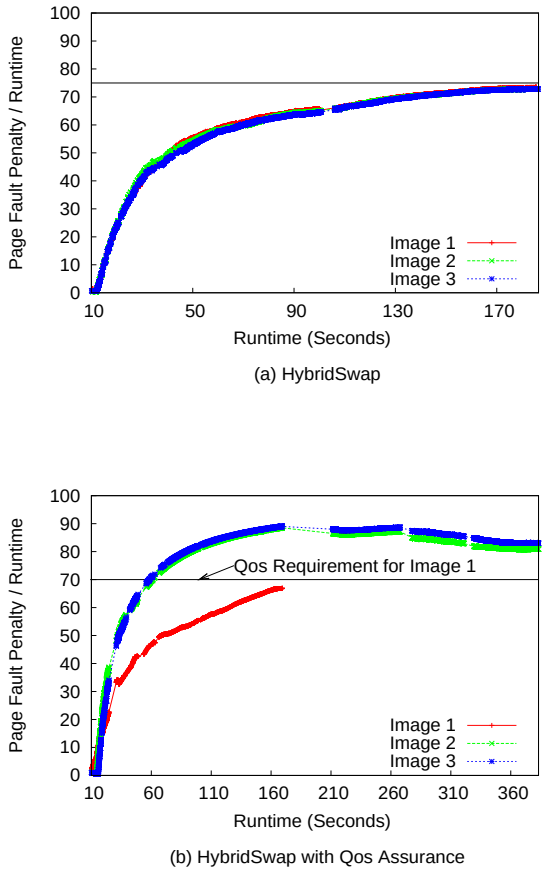

Figure 5.9: Running three instances of the Image benchmark with HybridSwap when a QoS requirement in terms of a bound on the ratio of the aggregate page fault penalty and (a) run time is not specified, or (b) is specified.

HybridSwap does keep this instance's ratio below the bound. Concurrently, the other two instances' ratios approach 90% and execution times are significantly extended. We note that Instance 1's execution time is reduced by only 10% and a tighter bound on the instance may not be realized because the use of the hard disk has already been highly optimized in HybridSwap and the relative performance advantage of the SSD is limited. A larger range for the QoS control to be effective would require a faster SSD (or SSD RAID) or larger memory. We note that the QoS mechanism could also

be useful for ameliorating the effects of spikes in swapping intensity, wherein at high swapping intensity traffic would be preferentially shifted to the SSD.

5.9 Runtime Overhead Analysis

The run time overhead of HybridSwap has two major components: the time for checking the access tables to determine the spatial locality of candidate sequences and the time for searching access tables for sequences to be prefetched when major page faults occur. Even though these operations are on critical I/O time, the time overhead is inconsequential because the time for operations on the in-memory data structure is at least two orders of magnitude less than disk latency and even SSD latency. In this section we quantitatively analyze the overhead of HybridSwap. Instead of directly measuring the overhead we measure the increase in program run time attributable to the sequence-related operations. More specifically, we measure the run times of three selected benchmarks (Image, CC, and Matrix) with RAID-Swap and compare them to the run times with RAID-Swap with the function module for these operations added. In the latter case, we ensure that the operations be carried out as they do in HybridSwap. The results are shown in Figure 5.10. The run time overhead is less than 1% on average for all the benchmarks.

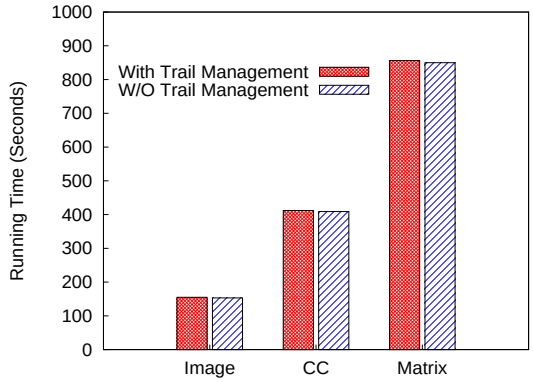

Figure 5.10: Run times of the Image, CC, and Matrix benchmarks with and without sequences-related operations invoked.

Chapter 6
Conclusions and Future Work

In this chapter, we first make a summary of this thesis, then we discuss the limitations in the design and evaluation of the proposed solutions. In the end, we suggest several directions for future work.

6.1 Contributions

We propose using the hard disk in addition to solid state drive to support virtual memory for hosting its swap space subject to two constraints. First, significant write traffic to the swap space should be directed to the hard disk to improve solid state drive's lifetime. Second, the performance of the hybrid swapping system should not be significantly less than that of a solid state drive-only system.

In our design of such a system we ensure that (1) temporal locality is effectively exploited so that the memory be fully utilized; (2) spatial locality is effectively exploited so that the disk does not become a performance bottleneck; (3) the swapping of pages to the solid state drive and to the hard disk is scheduled such that the solid state drive is used only to serve page faults when it can achieve a performance advantage over the disk; and, (4) the use of solid state drive and hard disk is coordinated so that the throughput potential of hard disk is exploited but its access-latency disadvantage is avoided.

We have implemented the proposed HybridSwap scheme in Linux for synergistic coupling of solid state drive and hard disk to serve as a memory extension. We experimentally compared HybridSwap to a solid state drive-only solution, and to a solid state drive/disk array, as swap devices in Linux using representative applications including key-value store for in-memory caching, image processing, and scientific

computations. Our evaluation shows that HybridSwap can reduce writes to the solid state drive by up to 40% with the system's performance comparable to that with pure solid state drive swapping.

6.2 Limitations and Future Work

Although this work has shown promising results with the proposed techniques to reduce solid state drive writes without comprosing performance, there are some limitations in this work and will be addressed in future work.

First, this work only considers about processes without the presence of large shared memory segments. In the current design, they simply put shared anonymous pages into solid state drives. For applications like database system, there might be lots of shared anonymous pages and our design needs to handle the case. Second, the benchmark of this work concentrates on scientific applications such as matrix inverse and correlation computation. We need to evaluate with some benchmarks in other application domains.

In future, first, we will take shared anonymous pages into our design to make our design could handle more general cases and test it with corresponding benchmarks. Second, we will evaluate our design with more common applications such as Word, video player. Third, we will incorporate the existing work of virtual memory making use of solid state drive only into our design such as FlashVM, SSDAlloc to find out how much improvement we can get together.

REFERENCES

[1] Alglib, a cross-platform numerical analysis and data processing library, 2012. http://www.alglib.net/.

[2] J. Axboe. Completely fair queueing (cfq) scheduler, 2010. http://en.wikipedia.org/wiki/CFQ.

[3] T. Coughlin and E. Grochowski Hard Disk Drive Capital Equipment and Technology Report, 2012. http://www.tomcoughlin.com/Techpapers/2012%20Capital%20Equipment%20Report%20Brochure%20021112.pdf.

[4] R. ARPACI-DUSSEAU Paging: Introduction http://pages.cs.wisc.edu/~remzi/OSFEP/vm-paging.pdf.

[5] M. Srinivasan and P. Saab. Flashcache: a general purpose writeback block cache for linux, 2011. https://github.com/facebook/flashcache.

[6] Imagemagick, 2012. http://www.imagemagick.org/script/index.php.

[7] J. Axboe. Noop scheduler, 2010. http://en.wikipedia.org/wiki/Noop.

[8] Memcached, 2011. http:/memcached.org/.

[9] V. Vleck, Thomas. Multics General Info and FAQ, 2012 http://www.multicians.org/general.html.

[10] Belady, L.A. A study of replacement algo-rithms for virtual storage computers *IBM Sys-tems J. 5*, 1966

[11] D. Bovet, M. Cesati. Understanding the Linux Kernel, Third Edition. *Oreilly*, 2005.

[12] A. Badam and V. S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *the 8th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2011.

[13] A. T. Campbell. A quality of service architecture. In *Ph.D Thesis, Computing Department, Lancaster University*, 1996.

[14] F. Chen, D. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2009.

[15] F. Chen, D. Koufaty, and X. Zhang. Hystor: Making the best use of solid state drives in high performance storage systems. *International Conference on Supercomputing*, 2011.

[16] Chase, J. S., H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system *ACM TOCS 12*, 1994

[17] Peter J. Denning. BEFORE MEMORY WAS VIRTUAL, 1996 `http://cs.gmu.edu/cne/pjd/PUBS/bvm.pdf`.

[18] Englander, Irv The architecture of computer hardware and systems software. *Wiley*, 2003

[19] Mel Gorman. Understanding the Linux Virtual Memory Manager. *Prentice Hall*, 2004

[20] M. M. Franceschini and L. Lastras-Montano. Improving read performance of phase change memories via write cancellation and write pausing. *The 16th*

IEEE International Symposium on High Performance Computer Architecture, 2010.

[21] R. F. Freitas and W. W. Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52, 2008.

[22] M. B. G. Soundararajan, V. Prabhakaran and T. Wobber. Extending ssd lifetimes with disk-based write caches. In *the 8th USENIX Conference on File and Storage Technologies*. USENIX, 2010.

[23] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing flash memory: Anomalies, observations and applications. In *the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE/ACM, 2009.

[24] L. M. Grupp, J. D. Davis, and S. Swanson. The bleak future of nand flash memory. In *the USENIX Conference on File and Storage Technologies*. USENIX, 2012.

[25] S. M. Hand. Self-paging in the nemesis operating system. In *the third symposium on Operating systems design and implementation*, 1999.

[26] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *SYSTOR'09: The Israeli Experimental Systems Conference*, 2009.

[27] T. Johnson and D. Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *International Conference on Very Large Data Bases*, 1994.

[28] S. Ko, S. Jun, Y. Ryu, O. Kwon, and K. Koh. A new linux swap system for flash memory storage devices. In *International Conference on Computational Sciences and its Applications*, 2008.

[29] S. Jiang and X. Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'02)*, 2002.

[30] T. Johnson and D. Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *International Conference on Very Large Data Bases(VLDB'94)*, 1994.

[31] Y. Smaragdakis and S. Kaplan. EELRU: Simple and Effective Adaptive Page Replacement. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'99)*, 1999.

[32] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. R. Nevill. Bit error rate in nand flash memories. *IEEE International Reliability Physics Symposium*, 2008.

[33] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for nvm+dram hybrid main memory. In *the 12th conference on Hot topics in operating systems*, 2009.

[34] L. Ray. SSD Flash drives enter the enterprise. *Silverton Consulting*, 2008

[35] M.Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear levelings. In *the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE/ACM, 2009.

[36] H. Payer, M. A. Sanvido, Z. Bandic, and C. M. Kirsch. Combo drive: Optimizing cost and performance in a heterogeneous storage device. *the 1st Workshop on integrating solid-state memory into the storage hierarchy*, 2009.

[37] T. Pritchett and M. Thottethodi. Sievestore: A highly-selective, ensembel-level disk cache for cost-performance. In *Proceeding of 37th International Symposium on Computer Architecture*. ACM, 2010.

[38] J. Ren and Q. Yang. I-cash:intelligently coupled array of ssd and hdd. *The 17th IEEE Symposium on High Performance Computer Architecture*, 2011.

[39] M. Saxena and M. M. Swift. Flashvm: Virtual memory management on flash. In *Proceeding of the 2010 USENIX Annual Technical Conference*. USENIX, 2010.

[40] S. W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G. R. Ganger. On multidimensional data and modern disks. In *the 4th USENIX Conference on File and Storage Technologies*. USENIX, 2005.

[41] Tanenbaum Distributed Operating Systems *Prentice-Hall*, 1995

[42] C. Wang, S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Eagelmann. Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines. *26th IEEE International Parallel and Distributed Processing Symposium*, 2012.

[43] William Stallings Operating Systems: Internals and Design Principles *Prentice Hall*, 2008

[44] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. *The 36th International Symposium on Computer Architecture*, 2009.

ABSTRACT

SYNERGISTICALLY COUPLING OF SOLID STATE DRIVES AND HARD DISKS FOR QOS-AWARE VIRTUAL MEMORY

by

KE LIU

May 2013

Advisor: Dr. Song Jiang

Major: Computer Engineering

Degree: Master of Science

With significant advantages in capacity, power consumption, and price, solid state disk (SSD) has good potential to be employed as an extension of dynamic random-access memory, such that applications with large working sets could run efficiently on a modestly configured system. While initial results reported in recent works show promising prospects for this use of SSD by incorporating it into the management of virtual memory, frequent writes from write-intensive programs could quickly wear out SSD, making the idea less practical.

This thesis makes four contributions towards solving this issue. First, we propose a scheme, *HybridSwap*, that integrates a hard disk with an SSD for virtual memory management, synergistically achieving the advantages of both. In addition, HybridSwap can constrain performance loss caused by swapping according to user-specified QoS requirements.

Second, We develop an efficient algorithm to record memory access history and to identify page access sequences and evaluate their locality. Using a history of page access patterns HybridSwap dynamically creates an out-of-memory virtual memory page layout on the swap space spanning the SSD and hard disk such that random

reads are served by SSD and sequential reads are asynchronously served by the hard disk with high efficiency.

Third, we build a QoS-assurance mechanism into HybridSwap to demonstrate the flexibility of the system in bounding the performance penalty due to swapping. It allows users to specify a bound on the program stall time due to page faults as a percentage of the program's total run time.

Forth, we have implemented HybridSwap in a recent Linux kernel, version 2.6.35.7. Our evaluation with representative benchmarks, such as Memcached for key-value store, and scientific programs from the ALGLIB cross-platform numerical analysis and data processing library, shows that the number of writes to SSD can be reduced by 40% with the system's performance comparable to that with pure SSD swapping, and can satisfy a swapping-related QoS requirement as long as the I/O resource is sufficient.

AUTOBIOGRAPHICAL STATEMENT

KE LIU

Ke Liu is a graduate student of Department of Electrical and Computer Engineering at Wayne State University. He received his B.S. degree in computer science and technology from National University of Defense and Technology, China in 2008, and M.S. degree in computer science and technology from Beijing University of Posts and Telecommunications, China in 2011.

His research interests include operating system, file and storage system, virtual memory, parallel I/O and MPI-IO library . He has published 2 paper, one in *IEEE International Symposium on Performance Analysis of Systems and Software* (2013, first author) and the other in *IEEE International Parallel and Distributed Processing Symposium* (2013, second author).